Task Activity Example

Delivery of diagnostics

Tue, Jul 28, 1998

Introduction

This note illustrates a measurement of task timing in an IRM as it executes a page application program whose function is to deliver task timing data. The page application is `PAGETASK`, and it is given a "keyboard interrupt," or click, to request a dump of all saved task activity up to that point. In order to capture the activity of the system that results from such a click, the request was issued two times in rapid succession, as the task diagnostics 4K-byte circular buffer typically holds only about ten 15 Hz cycles of diagnostic data. The diagnostics captured on the second click included the task activities relating to production of the desired task diagnostics data in response to the first click.

Scenario of test

In this example, the page was set up to list the results to the serial port of another node, which means that a series of setting messages, each of which includes one line of the 256 lines of text output, had to be built and sent to that node. Each line of text was 32 bytes. A Classic protocol setting message for this case adds up to a total message size of 46 bytes. With 256 such lines being delivered, the total to be transmitted to the network is 11776 bytes. This is more than the 9000-byte limit of UDP datagram support, so that fragmented datagrams must be sent to contain all this listing output. The following diagnostics will illustrate what happens.

When a page application is in danger of spending too much CPU time, it should respect the real-time nature of the front-end support and break up its activities by making calls to NextTask, which will allow any other pending tasks to execute. This example illustrates how this affects the job of building such a large listing output.

Task Timing Diagnostic Data

Each line of the following task timing data shows task name, followed by any pending events for that task, followed by the execution time of the task in ms, followed by the time of the start of task execution, shown in `hrmn:sc` and `ms` delay within the indicated 15 Hz cycle (range 00–14). The three cycles of task activities shown are separated by a blank line for clarity.

```
QMon   0000    .08 0911:45-02+ 0    Usual cycle start network cleanup
Updt   0000   2.25 0911:45-02+ 0    Usual data pool updating
QMon   0001    .3  0911:45-02+ 2
DTim   0000    .07 0911:45-02+ 2    Date/time update
Alrm   0000   1.12 0911:45-02+ 2    Usual alarm scan
Appl   0000   7.86 0911:45-02+ 4    Slice 1 in response to click
Alrm   0000    .05 0911:45-02+11
SDmp   0000    .06 0911:45-02+11
Serl   0000    .43 0911:45-02+12
Appl   0000  17.21 0911:45-02+12    Slices 2-4
Serl   0000    .14 0911:45-02+29
Appl   0000  11.77 0911:45-02+29    Slices 5-6
Cons   0000    .08 0911:45-02+41
Serv   0000    .09 0911:45-02+41
Appl   0000   5.87 0911:45-02+41    Slice 7
```

```
Serl   0000    .59 0911:45-02+47
Appl   0000   5.68 0911:45-02+48    Slice 8
Serl   0000    .33 0911:45-02+53
Appl   0000  16.73 0911:45-02+54    Slice 9-11

QMon   0000    .06 0911:45-03+ 4
Updt   0000   8.04 0911:45-03+ 4    Build first datagram (176 msgs)
Appl   0000   6.75 0911:45-03+12    Slice 12
QMon   0001  15.67 0911:45-03+19    Network cleanup after 1st datagram
Serl   0000    .45 0911:45-03+34
Appl   0000   6.05 0911:45-03+35    Slice 13
SNAP   0000    .2  0911:45-03+41
DTim   0000    .09 0911:45-03+41
Cons   0000    .07 0911:45-03+41
Serv   0000    .09 0911:45-03+41
Appl   0000   5.77 0911:45-03+41    Slice 14
Clas   0000    .19 0911:45-03+47
Alrm   0000   1.1  0911:45-03+47
Appl   0008   5.72 0911:45-03+48    Slice 15
Alrm   0000    .05 0911:45-03+54
Serl   0000    .13 0911:45-03+54
Appl   0008   5.52 0911:45-03+54    Slice 16 (last one)
Updt   0000   2.41 0911:45-03+60    Build second datagram (80 msgs)
QMon   0000    .07 0911:45-03+62
IDLE   0000   4.38 0911:45-03+62    Idle time at end of cycle

QMon   0000   3.89 0911:45-04+ 0    Extra network cleanup last datagram
Updt   0000   3.07 0911:45-04+ 4    Usual data pool update
Serl   0000    .4  0911:45-04+ 7    Serial port input processing
QMon   0001    .33 0911:45-04+ 7    (Everything back to normal)
DTim   0000    .1  0911:45-04+ 7
Alrm   0000   1.15 0911:45-04+ 8
Appl   0000    .14 0911:45-04+ 9
Alrm   0000    .04 0911:45-04+ 9
SDmp   0000    .06 0911:45-04+ 9
Updt   0000    .11 0911:45-04+ 9
QMon   0000    .06 0911:45-04+ 9
IDLE   0000    .11 0911:45-04+ 9
Serl   0000    .13 0911:45-04+ 9
IDLE   0000  30.43 0911:45-04+ 9
Cons   0000    .09 0911:45-04+40
Serv   0000    .09 0911:45-04+40
IDLE   0000  26.55 0911:45-04+40
```

Background of IRM task execution

The IRM operates in real-time, which means that it respects 15 Hz task activity. All tasks run at the same priority, because all jobs are normally expected to complete their work each cycle. In addition, time-slicing is not used, so that task switch can only occur at times that the currently running task voluntarily gives up the CPU. Any task that needs an extended time for execution should invoke `NextTask` from time to time in order to permit other tasks to run. A principal task for performing this logic is the `Appl` task, which invokes the current active page application. (Only one page application can be active at once; it is invoked by the system at 15 Hz after the data pool is filled, active requests are fulfilled, and the alarm scan has completed.) Via operator interaction, a page application is sometimes asked to perform an operation that may take a long time to complete—even more than a cycle. The active page application is viewed as a diagnostic, so that its absolute rigid adherence to 15 Hz execution is not demanded, but it should not prevent other tasks, such as `Updt` and `Alrm`, from maintaining real-time operation.

Analysis

The particular page application under study is written to break up its output of encoded diagnostics data by invoking `NextTask` every 16 lines of text. In this example, a total of 256 lines of output text are generated, so that means it invokes `NextTask` 16 times during the course of its encoding and queuing to the network the setting messages that carry the text to be sent via the network to the target listing node for local printing via its serial port. This activity is performed by the `Appl` task. According to the diagnostic record, each "slice" of Appl activity, encoding and queuing 16 lines of output, requires about 6 ms. When `Appl` calls `NextTask`, but it turns out that no other tasks are in the ready queue awaiting execution, `NextTask` immediately returns without switching to another task. This is why some slices require a significantly longer time than 6 ms. But the "quantum" of slice execution time is about 6 ms. We can deduce that encoding one line of text and queuing it to the network requires about 350 microseconds.

The `Updt` task refreshes the data pool early each cycle and fulfills any active data requests, but it is also used to flush any queued messages to the network. In this example, after 11 slices of `Appl` execution, a new cycle began, and the `Updt` task flushed all queued message to the network. This meant it built the entire first datagram of 176*46 = 8096 bytes and partitioned it into 6 fragments for queuing to the network hardware. Doing all this required 8 ms, which is 5 ms longer than the usual 3 ms time to update the data pool in that node. From the diagnostics, it appears that an extra execution of `Updt` required 2.4 ms to build the 80*46 = 3680 bytes and partition it into three fragments for queuing to the network hardware.

The `QMon` task does cleanup work after completion of network transmissions. In this case of setting messages, each allocated message block is freed upon completion of transmission to the network, and event marked by the network transmit interrupt. In this example, 15 ms of time was used by `QMon` when it sought to free up 176 message blocks. Later, after the second datagram transmission completed, it needed about 4 ms to free 80 message blocks.

Note that `Appl` task execution continued for longer than one cycle; in fact, it almost consumed all of the available time (that would otherwise be idle time) during two cycles. The first cycle of `Appl` activity used all available time, and only 4 ms of idle time remained in the second cycle. Also note that the start of task execution for the second cycle was delayed for 4 ms. This occurred because the cycle interrupt occurred while `Appl` was active running Slice 11, which required 6 ms. Switching to `QMon` was held up until `Appl` next invoked `NextTask`.

The `Alrm` task execution requires 1.1 ms in this node. It should run every cycle. For this example, `Alrm` ran much later than usual during the second cycle, but it did run, so that any device data residing in the data pool was checked for alarms.

One the third cycle, everything was back to normal, except for the `QMon` time to cleanup and free the 80 message blocks that comprised the second datagram. About 60 ms of idle time is available in a normal 66 ms cycle for this node.

Conclusion

This example shows how task execution can be broken up by a long-running page application by invoking `NextTask` so that the IRM's real-time character is not compromised. After studying these results, it might be advisable to reduce the time "quantum" used by this application. Perhaps `NextTask` should be called after processing only 8 lines of text, say. This would limit the delay of the second cycle's execution by no more than 3 ms, rather than up to 6 ms as demonstrated herein.